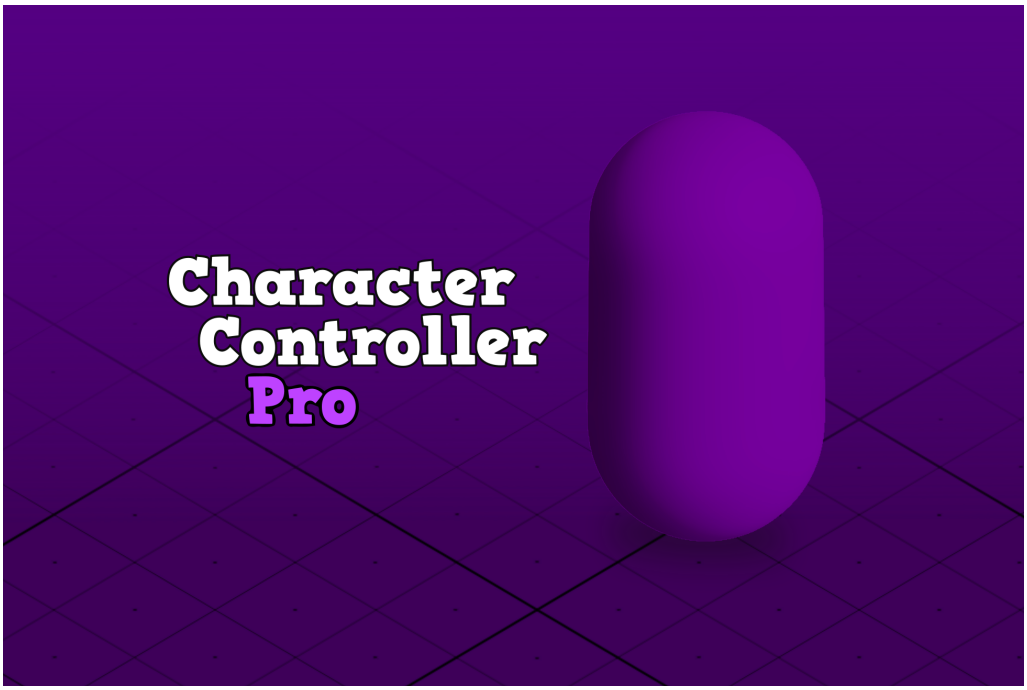
User Manual



Version 1.0.0

Lightbug

Email: lightbug14@gmail.com

Note: This document explains the basics of *"Character Controller Pro"* the best way possible. The document is written as a guide/manual with descriptions, definitions, images and side notes from me (Lightbug).

If **after** reading this manual you feel confused somehow or have questions about the package, please send me an email and i will gladly answer to your questions.

Also really important to mention, English is not my primary language, as always I'm doing my best with what I know.

Contents

Chapter 1

Introduction

1.1 Description

Character Controller Pro (CCP for short) is a 2D/3D Dynamic Character Controller that allows you to handle the movement, rotation and size of your character (among other things) in a precise way.

The main highlights of the package are listed below:

Hybrid approach This character controller is Dynamic by nature, this means that the character uses a collider and a dynamic rigidbody to detect collisions and to do movement. However, almost all of its actions are predicted/calculated in a "kinematic way". This was intended that way in order to combine the best of both worlds, that is, the preciseness of a kinematic character and the collision detection/rigidbodies interaction of a dynamic character.

Capsule body shape The character is modelled as a 2D/3D upright capsule.

2D and 3D Physics This package supports 2D and 3D Physics, meaning that your character will detect and interact with 2D and 3D Colliders as well.

2D and 3D Movement Every component available in the package was created with 2D/3D movement in mind. No behaviour is specific neither to 2D nor 3D space.

Rigidbodies interaction Since the character is dynamic it can interact with other rigidbodies, push them, get pushed by them, and gather information from the interaction.

Technical level This character controller not only gives you good results, but also technical information about the world the character is interacting with.

1.2 Content

Character Controller Pro is organized in three parts:

Core The main part of the package, it does the heavy lifting regarding collision detection, character information, movement, etc. If you want to extend the package or maybe you don't like the current implementation, you should use the core, it will give all the necessary tools to create your character on top.

Implementation Consist of a bunch of components that implement the functionality of the *Core*. These components cover many things like input handing, velocity based movement, graphics, animation, simple AI, Camera movement, etc. This part of the package acts as an example implementation of the *Core*, although it can be customize and extended if needed.

Demo Basically all the assets used for the making of the demo scenes, from materials and sprites, to pure data containers.

Although the implementation is encapsulated within the package, if you you need to handle things in a specific manner you can implement it all by yourself on top of the *Core*. This is why the *Core* and the *Implementation* are separated. Both have its own directories and are defined within its own namespace (see the API reference).

1.3 Asset version

The versioning scheme used for CCP is the following:

Major.Feature.Minor

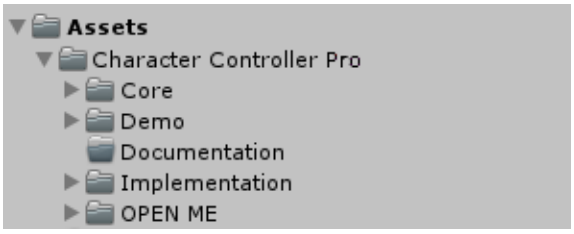


Figure 1.1: Project hierarchy right after importing the asset.

Minor Basically bug fixes, small changes, code improvements, etc. This update is always recommended, no risk at all.

Feature New features, small updates, a new fancy component, etc. Beware! It might break some things.

Major This update will break almost everything for sure, this means a new fresh start, new systems, new components, an overall change.

1.4 Unity version

CCP is developed and uploaded using Unity 2018.3.2, since this version presents a nice balance between modern features, comfort and usability. The package is compatible with 2018.3.2 or higher. If you notice some incompatibility with a newer version, please let me know.

I worth to mention that this package **might** work with older versions of Unity as well. It won't be uploaded to the asset store using any version older than 2012.3.2, though.

1.5 Setting up the project

1.5.1 Importing the package

Once the package has been imported your project view should look similar to what is shown in figure 1.1.

The first thing to do is to open the folder *"OPEN ME"*. In there you will find all the necessary material to put this package to work. In order to make CCP fully functional (at least for demo purposes) you need to adjust some Unity's settings.

The imported package contains a few presets[1], these are related mainly to "Lay-

[1]Predefined settings to use (and reuse) in a specific window type.

ers" and "Inputs". Apply these presets to your project, after doing this you will now visualize all the layers with their appropriate names, and the inputs buttons will be registered in the input manager.

The layers and inputs settings are necessary initially for demo purposes only. If you want to use CCP in your own project by your own rules you can redefine these settings to your liking.

What it is no optional is the execution order of the scripts. In the package there is an image inside a folder called *Execution Order*, which shows the Execution Order settings at the moment of release (since these settings cannot be saved as presets). Make sure to use that order in your project (the integer number doesn't care, what matters is the order, what's on top of what).

1.5.2 Using the package

The recommended way to use the package (and every package out there) is by working on a separated path (outside *"Character Controller Pro"*). This is useful is you need to use, extend or customize the asset in any way.

1.5.3 Updating the package

To update the package (regardless of the version) is recommended to delete the *"Character Controller Pro"* folder in its entirely, then import the updated version. Remember always to put your own work outside this folder.

Chapter 2

The core

This chapter explains how the main parts of the Core works, especially the ones related to movement. A brief description of the relevant components are described below (for more information refer to the API reference).

2.1 Components

The core consist of the following main components:

Character motor The main character component. It takes care of all the important actions (movement, rotation, size, among other things). It also holds all the collision information and is responsible for triggering events. Basically the entire package refer in one way or another to this component.

Character graphics This component is responsible for controlling the transform properties used by the character *graphics*.

Character debug This component is used for debug purposes, mainly to print information on screen about the collision flags, values and triggered events.

2.2 Body

2.2.1 Foot position

The foot position is considered as the origin, a point of reference for everything. As is normally for character controllers, the origin is assumed to be the bottom most point of the character body. In this case is the bottom point of the capsule.

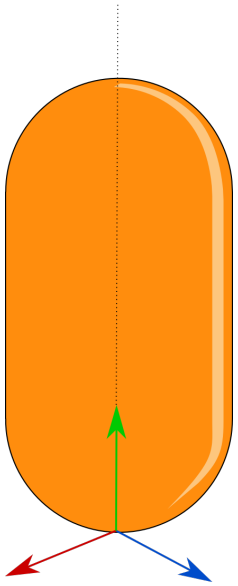


Figure 2.1

2.2.2 Orientation

The character is modeled as an upright capsule. This means that the capsule height vector[1] and the up vector of the character transform will have always the same direction vector (see figure 2.1).

This capsule is not visible, any real character will need a graphics element with a renderer to show the mesh or sprite on screen.

2.2.3 Size

Since this is a capsule-based character we need only to define the radius and the height to fully describe the size of the character. The *CharacterMotor* will consider the character dimensions as:

$$Width = 2 \times Collider.Radius \tag{2.1}$$

$$Height = Collider.Height \tag{2.2}$$

The Width and Height values are represented using via the "body size" (a *Vector2*). The figure 2.2 shows the capsule width and height.

[1] Vector defined from the bottom sphere to the top sphere

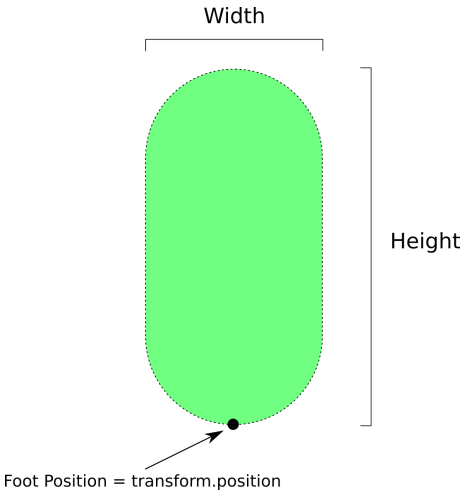


Figure 2.2: The character body.

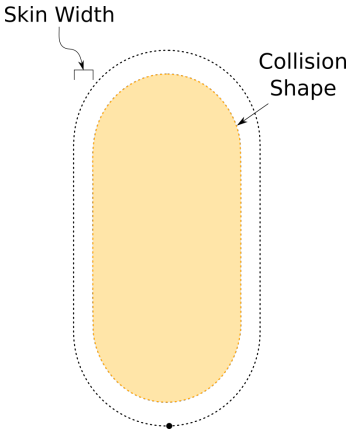


Figure 2.3: The collision shape of the character.

2.2.4 Collision shape

Even though this is a dynamic character controller, the *CharacterMotor* is still doing physics queries to detect collisions. Because of this an effective collision shape was defined (different from the character body) using the width, height and skin width (See figure 2.3).

2.3 Character behaviour

The *CharacterMotor* component follows a few simple rules that needs to be understood in order to use it. These rules are related to teleportation, size, position and

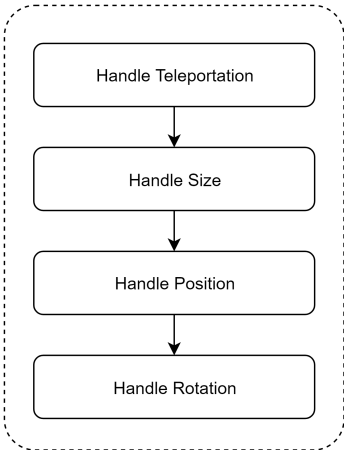


Figure 2.4: CharacterMotor internal actions.

rotation. All the actions required from the *CharacterMotor* have a deferred nature, this means that they are not executed immediately. All the changes are applied when the *CharacterMotor* needs to.

The main properties to look for are shown in figure 2.5. The *ChracterMotor* offers a few public methods and properties that allow you to define these properties.

The following sections explain how all these individual elements work internally.

2.3.1 Teleportation

If the character needs to change instantly its position and/or rotation (ignoring the collision detection algorithm) then the Teleportation method should be used.

2.3.2 Size

Internally the size of the character is stored in a Vector2 variable called `bodySize`. The x component represents the width and the y component represents the height. The initial size will be assigned based on the capsule collider parameters and some internal constants.

The size can be modified externally by a script (using the `SetBodySize` method), passing any value possible as the desired size. Since the character may or may not fit in a given space, the size must be internally checked beforehand by the *CharacterMotor*, thus defining the final character size.

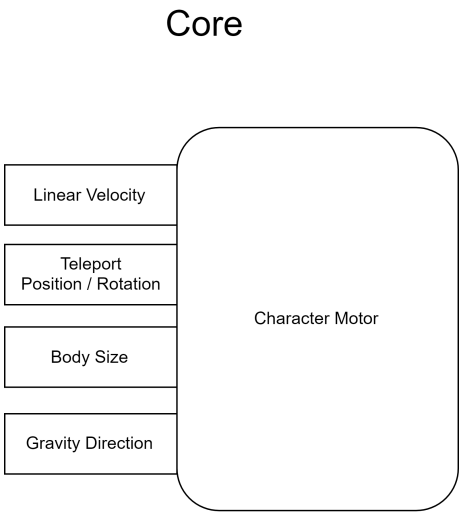


Figure 2.5: CharacterMotor internal actions.

2.3.3 Orientation

The orientation is completely managed by the *CharacterMotor* component. Any external change to the character rotation will be overwritten. There are two ways to describe a normal character rotation:

Gravity The current gravity direction will define the character up vector as its opposite.

Yaw This is the rotation motion around the local up axis.

The character is capable of changing its rotation values by using the gravity information. This is possible by using the gravity direction and graviy modes from the CharacterMotor component. For example, in figure 2.6 the character is align itself using the gravity vector.

On the other hand, the character cannot do yaw rotation at all, instead it has a *forward direction* vector, which can be defined and used for the user at will (see figure 2.7). This vector will be properly rotated internally by the *CharacterMotor* (gravity shifting scenarios and dynamic ground motion).

It is important to mention that the forward direction vector will always be projected onto the plane formed by the character up direction. If a random vector is defined as forward direction, this will be projected onto this plane.

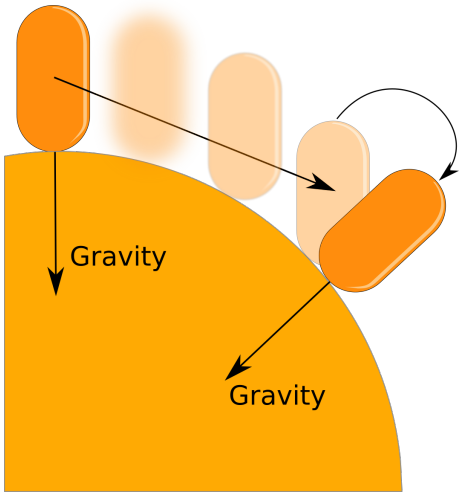


Figure 2.6: The character orientation being changed in a gravity shifting scenario (a planet in this case).

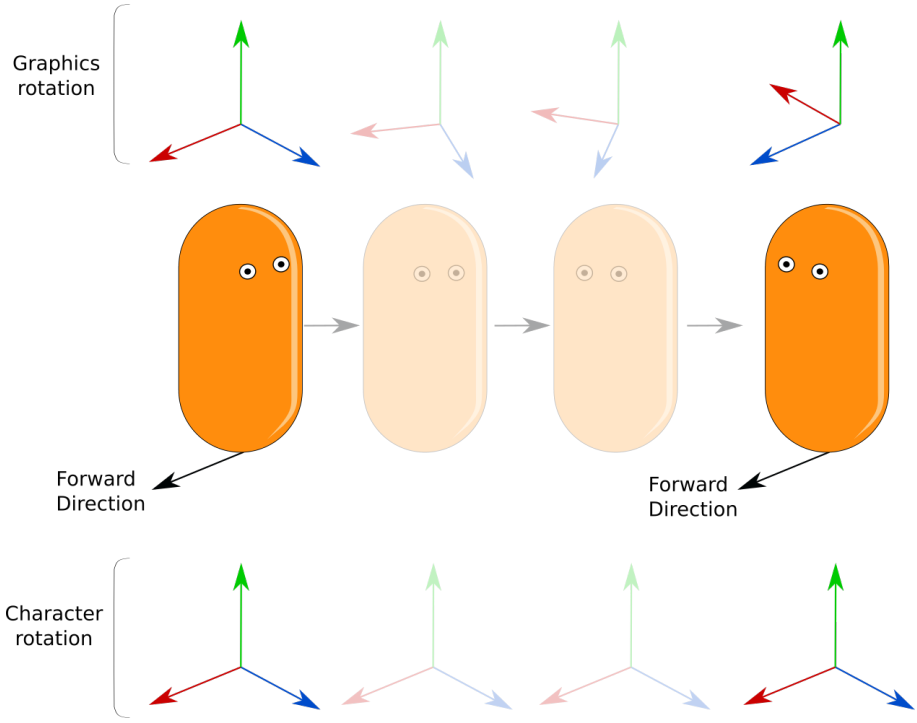


Figure 2.7: The graphics doing yaw motion towards the forward direction vector. The character rotation is not modified whatsoever.

2.3.4 Position

The change in position related to a *LinearVelocity* field. The necessary displacement is calculated based on the current linear velocity value. This movement will take into account collision detection. The *LinearVelocity* vector needs to be set externally by a script.

The movement algorithm is based on the classic *Collide & Slide* algorithm. Although is not necessary to do a bunch of collision test in order to avoid the character to pass through other colliders (since this is a rigidbody based character controller), these test are still performed. This is because the character gathers information from the environment (see section 2.4) and also can predict its movement before hand, which is really useful.

Grounded state

The movement can be classified in *Grounded* and *Not Grounded*, depending on the current grounded state. Both type of motion have its differences, the not grounded movement only performs the basic collide and slide action, while the grounded movement does the same but also includes all the available features (step detection, edge detection, edge compensation, etc).

Grounded* to *Not Grounded To make the transition you must call the `ForceNotGrounded` method. If you are using gravity in your script remember to apply a positive vertical velocity (towards the character up vector), otherwise this will return to the grounded state the next frame.

Not Grounded* to *Grounded In this case you must apply a negative vertical velocity (towards the character up vector)

Stability

A character is stable when it is grounded and the *stable slope angle* (see the API reference) is less than or equal to the *slope limit* value.

```
1     stable = IsGrounded && ( stableSlopeAngle <= slopeLimit );
2
```

This angle should not be confused with the *slope contact angle*, obtained directly from the collision test (See figure 2.8).

The concept of stability is used internally by the *CharacterMotor* to detect steps, edges, do ground probing, and so on. By itself it will not produce any type of movement, but it can be used for creating custom behaviours. For example, in the included implementation, a grounded and unstable character will slide down from

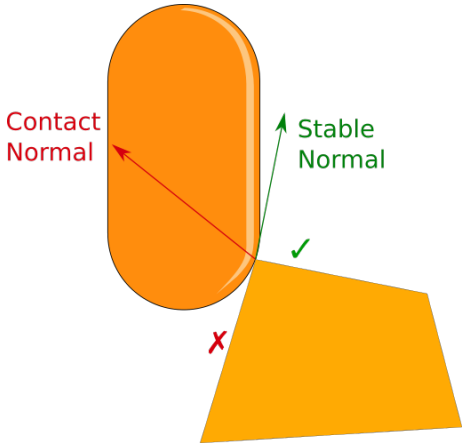


Figure 2.8: *Stable normal* vs *contact normal*.

the slope it is currently standing on. For example, referring back to figure 2.8, the character will not fall since the stable slope angle (from the stable normal) is allowed.

Velocity Projection

The *LinearVelocity* vector provides the information needed to move the character from point A to point B. However, this vector is not used directly to calculate the actual displacement, since it may not follow the *CharacterMotor* internal movement rules. This can be a problem only when the character is grounded.

In order to use a valid displacement vector, the linear velocity must be projected onto the current slope plane. The *CharacterMotor* will take the linear velocity field and do the projection maintaining its magnitude. This can be seen much more clearly in figure 2.9.

Important: the displacement vector will be created maintaining its initial magnitude. This basically means that the character will always move at its input velocity magnitude, regardless of the slope angle. This can be seen in figure 2.9.

Slopes handling

The character can walk onto any slopes whose angle is less than the slope limit value. Prior to the movement calculation a displacement vector is created (based on the linear velocity) and subsequently modified in the process. The collide and slide algorithm will iterate over and over until the displacement magnitude is less that the minimum movement amount constant, or the number of iterations performed exceeds the maximum number of slide iterations.

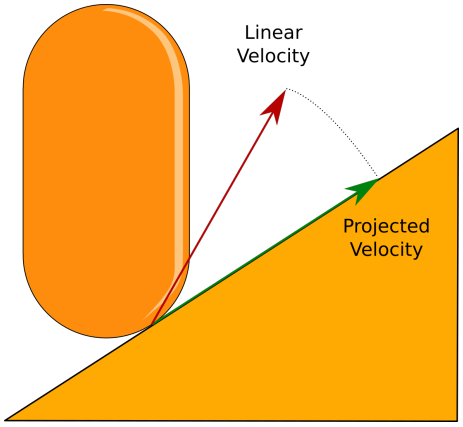


Figure 2.9: The *LinearVelocity* projected onto a stable slope.

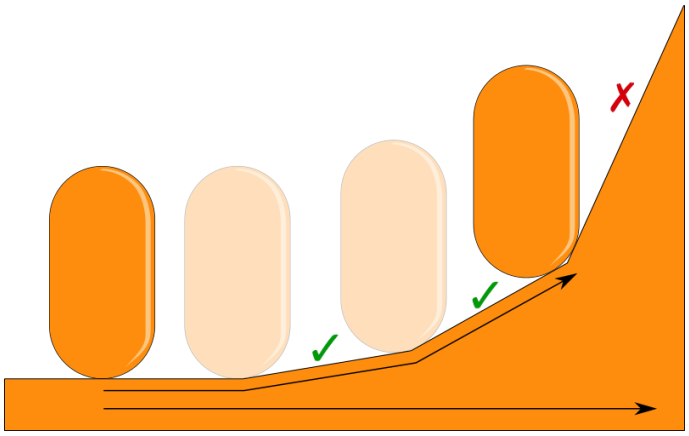
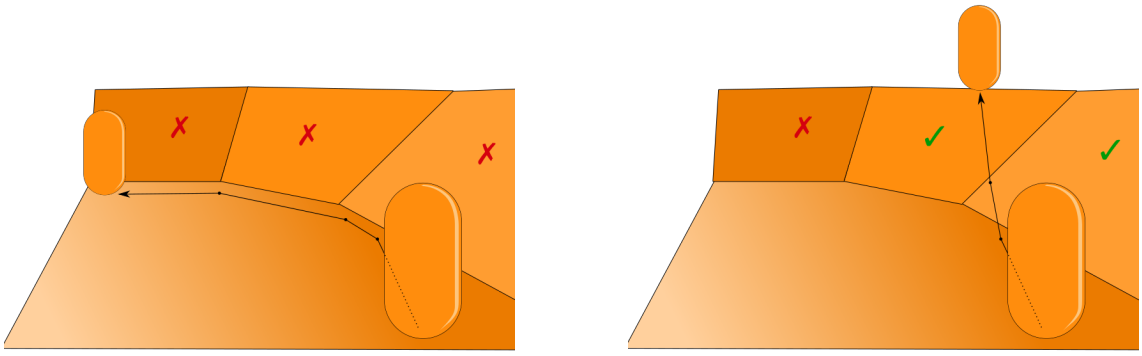


Figure 2.10: The character climbing up a slope.

If the encountered slope is allowed the character will walk onto it, modifying the displacement vector. if not the slope will be considered as an invisible wall. See figure 2.10 and figure 2.11 for more detail.



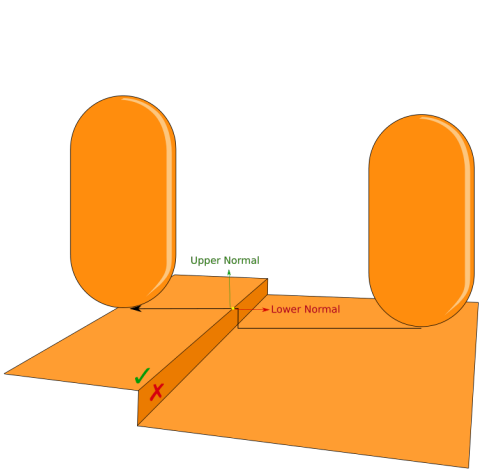
(a) Unallowed slopes. (b) Allowed slopes.

Figure 2.11: Deflection of the displacement vector.

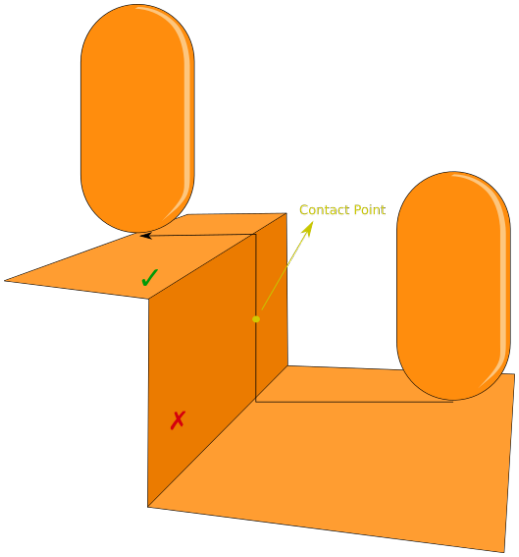
Steps handling

Step Up The character is able to walk over steps of any height (although it may looks weird in some situations). The basic rules to trigger the step up functionality are simple. During the collide and slide stage, if the character hits a wall (close to 90 degrees ± a tolerance) or if hits an edge (See figure 2.12), it will perform a "step up". After the process, if the ground surface is unstable the step up operation will be ignored (reverting the changes), and the step will be considered an invisible wall (same as an unallowed slope), recalculating the displacement vector.

To determine the stability of the new potential surface an edge detector is used . This allows to measure the upper and lower normal. Depending on the situation the chosen normal will be one or the other. The edge detection is independent from the orientation.



(a) Detecting an edge.



(b) Detecting a wall.

Figure 2.12: Step detection examples.

Step Down The step down algorithm (also known as ground clamping or ground snapping) will force the character to the ground if the ground probing distance[2] is less than the `StepDownDistance`. This is useful to prevent the character to be launched off a slope or step.

[2]Distance between the character foot position and the closest ground point measured.

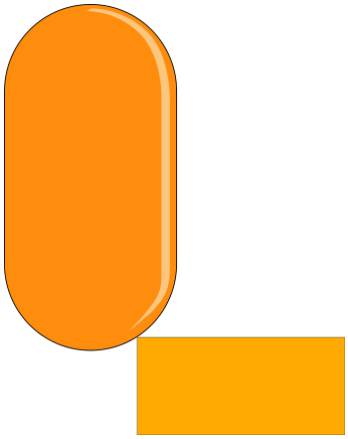
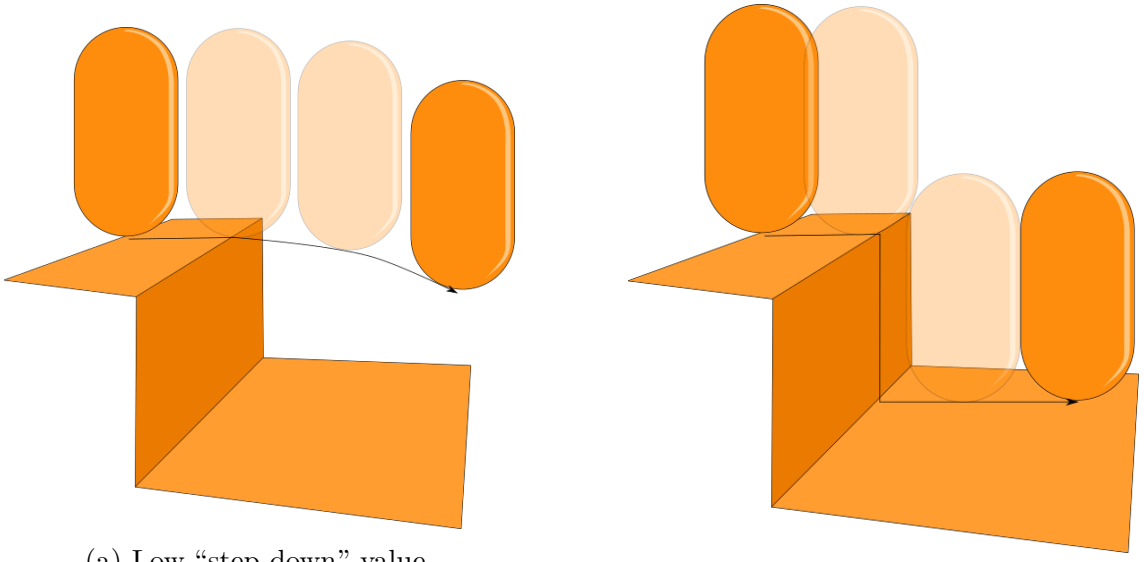


Figure 2.14: A character with *edge compensation* disabled.



(a) Low “step down” value.

(b) High “step down” value.

Figure 2.13: Step Down examples.

Edge Compensation

Normally, when a character is standing on an edge it collision shape will make contact with it. This is shown in figure 2.14;

Sometimes detecting the ground using a box shape can be the best option, especially for platform games. The character motor include a *edge compensation* feature, simulating a box shape for ground detection in edges.

This feature works only on edges, for slopes you will get the same results as

(a) On an edge.

(b) On a slope (no effect).

Figure 2.15: *Edge compensation* examples.

before (with edge compensation disabled), using the advantages of the box and the capsule shape all together.

Dynamic ground

If the character is standing on a valid dynamic ground[3] and this object is moved or rotated, the character will act properly to move/rotate along with it, always following its own orientation rules. This is shown in figure 2.16, see how the character maintains its rotation while it moves accordingly with the platform.

2.4 Collision information

A very import aspect of every character controller is the information this provides to the user. This information is really important to set your basic movement rules, like for example detect if the character is grounded or not, use the ground normal, gather the current ground information, etc.

CCP offers this information in form of public properties. In order to access them you need a reference to the *CharacterMotor* component and you are good to go.

For more information about the collision information see the API reference.

[3]Kinematic rigidbody moved and/or rotated using kinematic motion, either by script or via animation (with "Animate Physics" enabled).

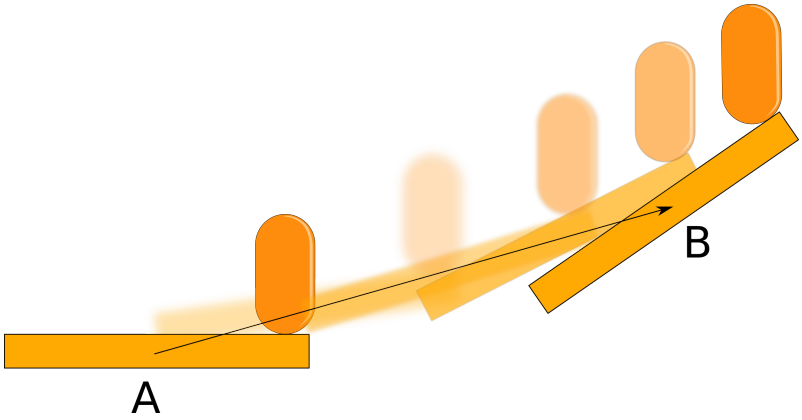


Figure 2.16: A 2D representation of a character standing on top of a kinematic platform (dynamic ground) that moves from point A to point B.

2.5 Collision events

A collision event is just a delegate event that is called whenever a particular situation happened (in this case related exclusively to collisions). When a specific condition is met, the related event will be called by the *CharacterMotor*, therefore calling any method subscribed to it.

The package include a number of collision events (Core and Implementation as well), if you want to look at them please refer to the API reference. All the events names start with the word "On".

If you want to see all these events in action, or simply see a code example please check the *CharacterDebug.cs* script. It contains a few methods subscribed to all of the available events. You will notice that every delegate event has its own signature, this is because they are passing information along when the event is called.

For example when the *OnHeadHit* event is called a copy of the *CollisionInfo* structure is passed as an argument, so you can get info from the collision itself (for example the *contactNormal*).

2.6 Rigidbodies interaction

2.6.1 Push

The character can push other dynamic rigidbodies by colliding with them. The resulting movement will be determine by the interaction and the rigidbodies parameters (relative velocity, mass, drag, etc). This means that the character will push more easily lighter rigidbodies. Since the velocity of the rigidbody is managed

by script, in order to increase (or decrease) the push force you must increase (or decrease) the character rigidbody mass.

It is important to assign the corresponding rigidbodies layers to the *CharacterMotor* layer mask in order to produce faithful results. The character must ignore these bodies when detecting collisions, otherwise the interactions will not be correct.

2.6.2 Weight

If the character is standing over a dynamic rigidbody this will apply a force to it (at the contact point) proportional to the rigidbody mass.

2.6.3 Collision response

If another dynamic rigidbody hits the character, this will receive a "contact velocity" due to the collision. Since the velocity is fully scripted, the script involved in the movement will need to know when and how the collision happened in order to react to it. This is handle by an event that is triggered every time a collision happens, passing the "contact velocity" as an argument.

This allowed rigidbodies to create this type of contact need to be tagged properly. See the *contactRigidbodiesTag* field.

2.7 Character graphics

Any character in a real-game scenario can be separated into two parts, a physics component and a graphics component, both working together to produce the result we want. The physics component is referred as the *Character Body* (or simply the "character"), and the graphics component as the *Character Graphics*.

Character body This is the actual "character", it does calculate everything in order to detect collisions, set flags, trigger collision events, and so on. This object is the one with the *CharacterMotor* component in it.

Character graphics It corresponds to what you actually see on the screen. It includes all the extra components that don't relate directly to the collision body, things like renderers, animations, particles effects, etc. This object is the one with the *CharacterGraphics* component in it (See section 2.7 for more info).

Decoupling the graphics from the collision body can be very advantageous, since we can handle the position and rotation independently from the actual character

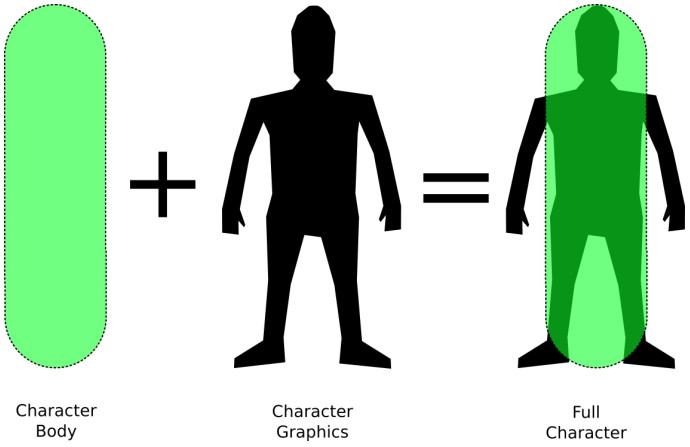


Figure 2.17

object. For instance we can produce any rotation we want (a common example is the Yaw motion).

The *CharacterGraphics* component is assigned to a *graphics object* (hence its name), taking care of its transform properties, such as position, rotation and scale changes. If you notice the *CharacterGraphics* does not do any graphics related tasks, such as managing animations, renderering, shaders, etc. Instead it is used to define what is graphics inside the character hierarchy. This means that the *graphics object* doesn't have to necessarily include all the graphics components directly in it (for instance these elements can be added to its children instead).

Think of the "graphics object" as the root of every graphics related object of your character. If it does graphics it must be placed as a child of this root object.

The *graphics object* is treated as a separated object from the character. Once the game is running (play mode), the graphics object and the character object will be separated from each other.

2.8 Setup of a basic character

2.8.1 Manually

Adding a character to the scene

Adding and setting up your own character from zero is really simple, follow these steps:

1. Add an empty *GameObject* to the scene. This will be the character GameObject, so let's call it "Character".

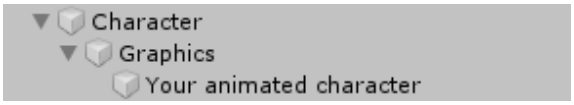


Figure 2.18: Automatic template for a character.

2. Add a *CharacterMotor2D* component to the empty object if you want the character to interact with 2D Physics. If the character should interact with 3D Physics a *CharacterMotor3D* will need to be added. Any of these action will automatically add a CapsuleCollider (2D or 3D) and a Rigidbody (2D/3D) component.

3. Adjust the collider dimensions (see figure 2.2).

Adding graphics

1. Create a child empty object for "Character". Since this corresponds to the graphics part let's call it "Graphics".

2. Add a *CharacterGraphics* component to it.

3. Add your animated model object as a child of "Graphics". This object will contains the actual renderers, animation components, animation scripts, etc.

2.8.2 Using the menu

Another way to add a character to the scene (and skip all the previous steps) is by going to the hierarchy window *"Create/Character Controller Pro/2D Character"* or *"Create/Character Controller Pro/3D Character"*. After this you will see a new empty character on the screen with the hierarchy as shown in figure 2.18.

2.9 Implementing the core

The one thing to do next is to use these concepts in a practical way. Usually a character controller[4] is needed to produce some basic moments. In order to create this type of component first you need to know all about the public fields and methods the CharacterMotor offers to you (see the API reference for more detail). Once you are familiarized with it you can start to create your own scripts around this. These scripts must hold a reference to the core components (or at least the ones you're

[4]High level component that interacts with the low level character components in order to produce movement.

going to use). Once this is done the way is clear for you to define (within the scope of CCP) your own vision for your character.

The implementation contained within the package is one big example of this working (and much more). If you want know all about the implementation please read chapter 3.

Chapter 3

The implementation

3.1 Description

The *Implementation* consist of set of scripts that implement the functionalities of the *Core* (see figure 3.1). Basically starting from a low level concept (the core) to a high level concept (a character system with game components).

This part of the package contains some components that will help you right away, handling movement, environment settings, animation, particles, moving/rotating platforms, camera controllers, AI behaviours, and so on. The main goal of this part is to define rules for character, starting from a low level concept (the core) to a high level concept (a character system with game components).

This implementation can me modified to fit a particular game style (for example an RTS game, a 2D platformer, a top-down shooter, etc.). It has been created as a general purpose implementation.

3.2 Character state controller

The *CharacterStateController* is sort of the main component from the *Implementation* part. It defines a code structure based on the concept of a "state".

There are a few things that can happen inside a state logic (depending on the set of rules chosen), such as action detection (from a input device or an AI behaviour), material interaction, movement, rotation, and so on. So, it is important to handle all these aspects in an orderly manner.

Apart from handling the state logic, this component also hold all the important data the states are going to share between each other.

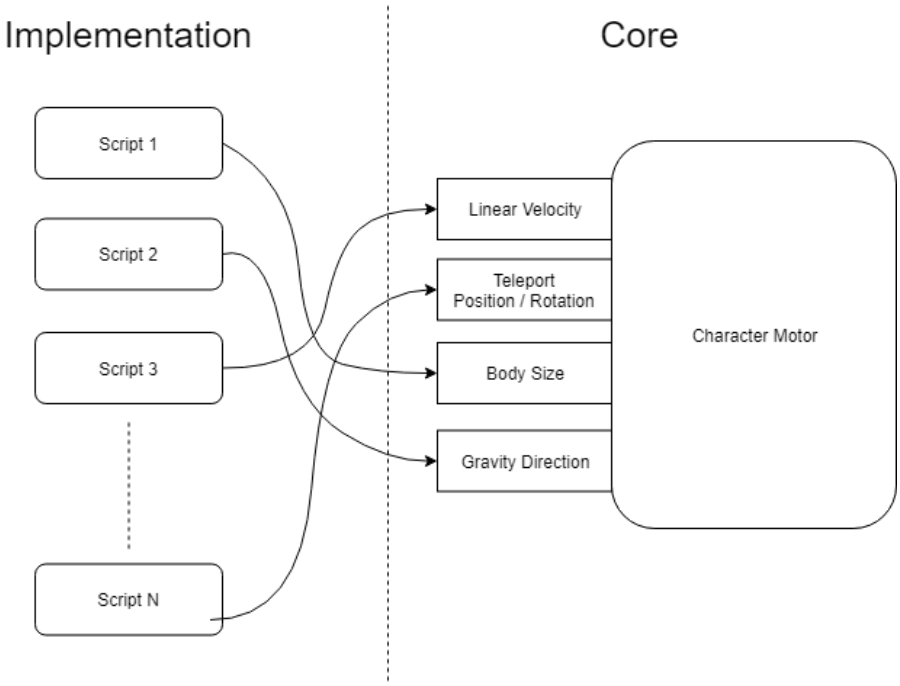


Figure 3.1: Core and Implementation.

3.2.1 Finite State Machine

This component implements a "finite state machine" (FSM). This machine is responsible for managing and executing all the valid states of the character and the transitions between the states.

A simple and crude representation of the structure of a state machine is shown in the example of figure 3.2.

The main loop cycle of the *CharacterStateController* component is shown in figure 3.3.

3.2.2 Movement direction

The *CharacterStateController* component include a simple property that can be used by the states to obtain a movement direction vector, based on the current input and a "reference". This direction is called *input movement reference*.

The *input movement reference* calculation is expressed in the block diagram of figure 3.4.

Important: This vector is just the result of super basic algebra between input data and a transform component. All the information needed to create it can be obtained from inside the state. So, it's ok if you don't want to use it for your own

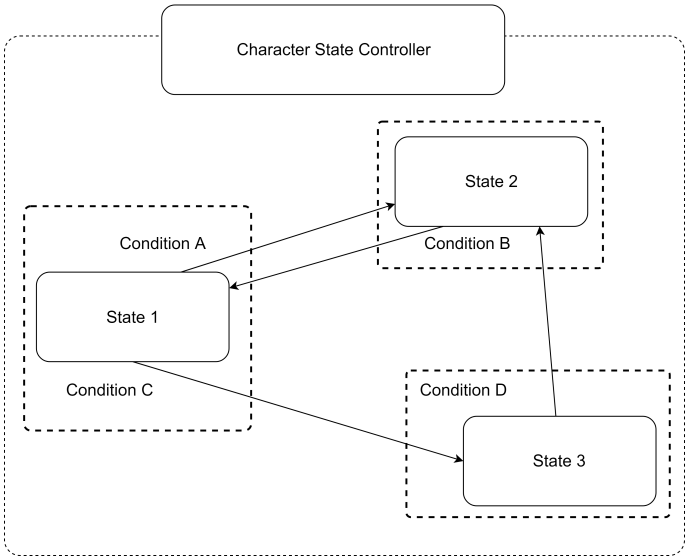


Figure 3.2: A representation of the state machine.

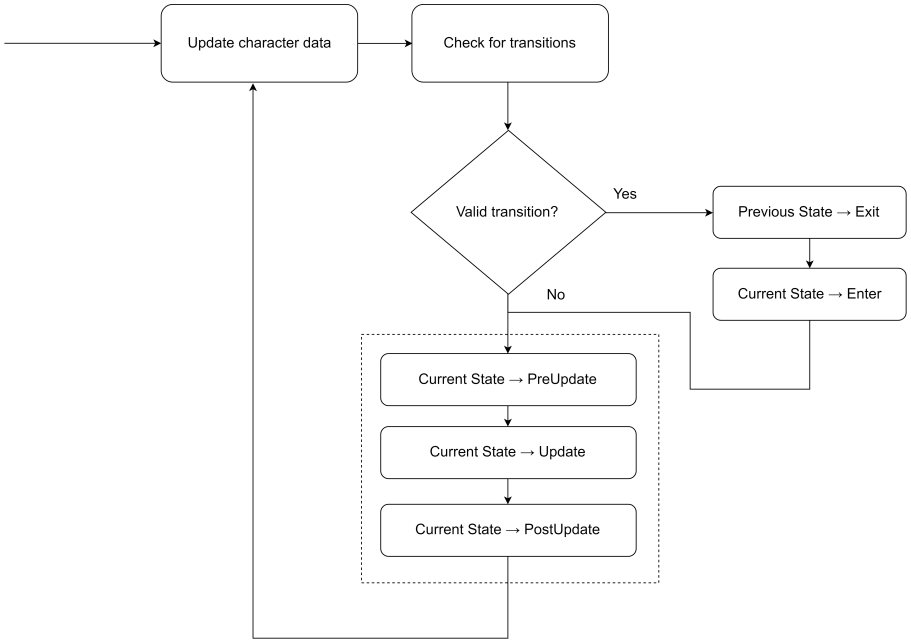


Figure 3.3: Main loop of the *CharacterStateController* component.

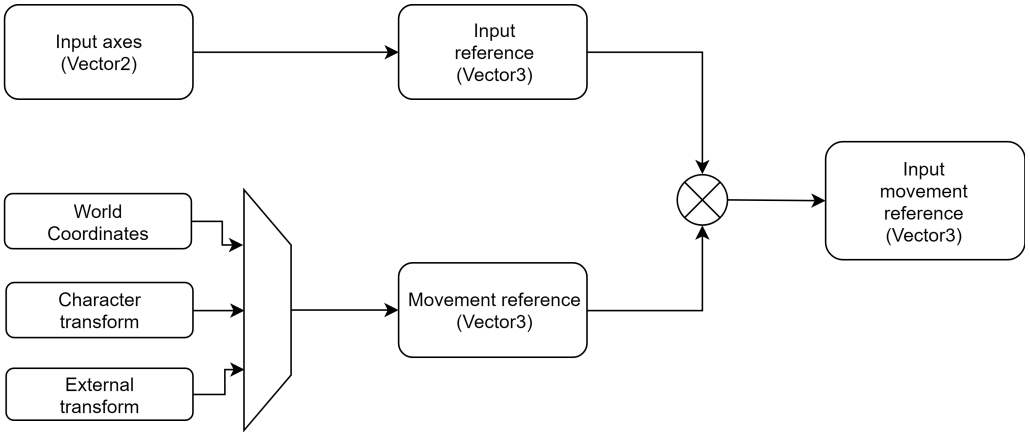


Figure 3.4

state logic, just know that it is there.

Input reference

The input reference is defined as a vector created exclusively by input actions (AI or Human), in this case by the *input axes* information. By default the *input axes* are defined as a *Vector2* that contains the *Horizontal* and *Vertical* axes values.

$$inputReference = \begin{pmatrix} inputAxes.x \\ 0 \\ inputAxes.y \end{pmatrix} \tag{3.1}$$

Since we are in 3D (general case), the z component of the input reference corresponds to the y component of the input axes. This is due to the fact that the vertical component (character up direction) is used for jumping, gravity, etc. So, we transformed the y component of the input axes (*Vertical* axis by default) into a forward direction. For 2D this component is obviously zero.

Movement reference

A *movement reference* is defined as a set of orthonormal vectors (think of the typical "right, up, forward").

There are three types of references available:

World The reference uses the world coordinates, this means that the *right*, *up* and *forward* directions are equals to *Vector3.right*, *Vector3.up* and *Vector3.forward* respectively.

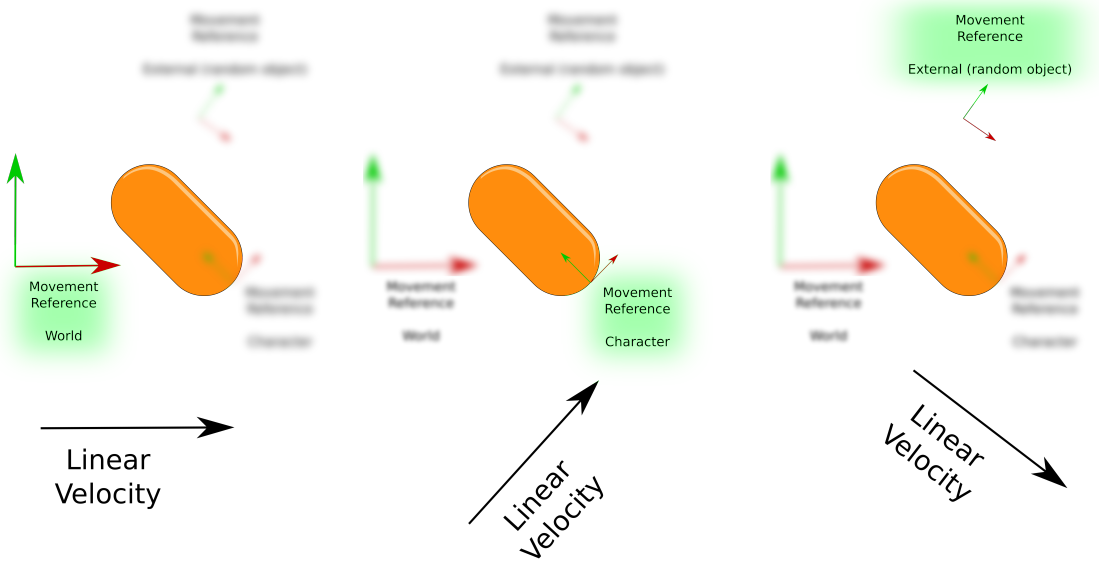


Figure 3.5

Character The reference uses the own character transform (right, up and forward) to update its components.

Character The reference uses an external transform to update its components.

Input movement reference

By combining the input reference with the movement reference (multiplication) it is possible to create sort of a mix between inputs and movement reference.

$$inputMovementReference = \begin{pmatrix} inputAxes.x \times movementReference.x \\ 0 \\ inputAxes.y \times movementReference.z \end{pmatrix} \quad (3.2)$$

To better clarify this concept see figure 3.5. There are three cases (World, Character and external), in which the same input axes vector (in this case the "right" action) is applied to each one of them. The figure shows three very different results, depending on the movement reference used.

Once the *InputMovementReference* vector has been defined, it only remains to multiply it by the speed required.

The *InputMovementReference* vector is updated before the states main loop. All the character states can have access to the *InputMovementReference*, and perform its own calculations to determine the velocity.

3.2.3 Materials

Consist of pure data (*ScriptableObjects* in this case) that contains properties for different materials. These materials are defined in the context of the *CharacterStateController*, shareable by all the states.

These materials can be used to affect the resulting character movement. The type of material affecting the character will depend on the grounded state, the material properties, and finally the movement properties of the character state.

For instance, in the *NormalMovement* state, the material parameters are used to modify the velocity sent to the *CharacterMotor*.

A material can be a **volume** or a **surface**:

Volume All the space around the character (not grounded parameters). This space can be made of air, water, jelly, and so on.

Surface The ground the character is standing on (grounded parameters). A surface can be ice, mud, grass, etc.

There are parameters that can be configured for both volumes and surfaces. These parameters are related to the amount of grounded control, not grounded control, gravity and speed modifiers.

Any material without a proper tag on it will be considered as a "default material".

3.3 Character state

3.3.1 State behaviour

A state is the main piece of code that the state machine executes. Every state presents its own behaviour, and can be implemented through its abstracts and virtual methods. The state controller will call them when they are needed, so don't worry about the execution order (see the API reference to know what methods are available):

You need to override the method you want to define a specific behaviour. For instance, if you want to create your own exit behaviour you can do something like this:

```
1  // YourCustomState.cs ------------------------------
2
3  public override void ExitBehaviour(float dt)
4  {
5    // Your code ...
6  }
7
```

3.3.2 State creation

If you want to create your own states you can do so manually by creating your own class. It's mandatory that it derives from the *CharacterState* class.

Another way is by simply right clicking on the project view (choose the path of your choice), then go to *"Create/Character Controller Pro/Implementation/Character State"*. This will create a template with the basics already included.

3.3.3 Transitions

This is an important concept to understand. A transition is evaluated in two places, in the "from state"[1] and in the "to state"[2]. These transitions are called Exit transition and Enter transition.

Exit transition This is the part of the transition that is evaluated first, when trying to exit the current state (hence its name).

Enter transition This is the part of the transition that is evaluated secondly, when trying to enter the next state (hence its name).

Why do transitions using this approach? Well, sometimes the information needed to decide if a transition is successful or not is shared between two or more states, but sometimes it doesn't. Since we can't be sure that one state has the total control over one particular transition, dividing the condition test sounds like the right thing to do.

[1] State where the transition originates.
[2] State where the control flow goes to.

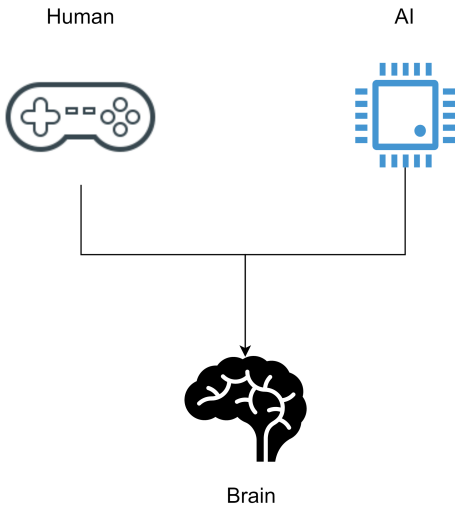


Figure 3.6: Representation of the Human, the AI and the brain.

Example Let's say you want to implement a transition to a *JetPack* state. In the *Normal* state, to activate the jet pack only a specific button press is needed. This state doesn't have any information about the *JetPack* state whatsoever (and it shouldn't). For the *JetPack* state is another story.
The *JetPack* state must check if everything is ok, before allowing the transition to happen. For instance, if the character jet pack doesn't have any fuel, this should not be active. In the transition code inside the *JetPack* state we can make sure that the fuel is enough.

3.4 Character brain

The character brain is a component responsible to handle all the character actions. These actions can be triggered either from a human player or the AI.

All the available actions are predefined in a structure and updated by the "Brain" component at runtime. This approach create a level of abstraction between the inputs (GetKey, GetButton, etc.) and the character actions themselves (jump, move forward, etc.). This is represented in figure 3.6.

To select one mode or the other simply click on the buttons in the inspector. It should look like figure 3.7.

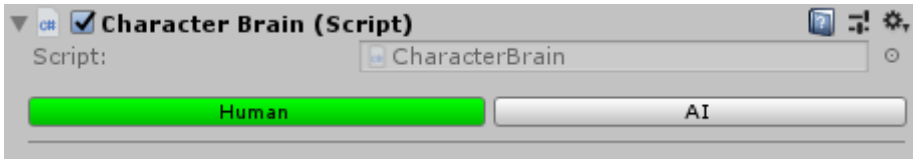


Figure 3.7: Brain modes in the inspector.

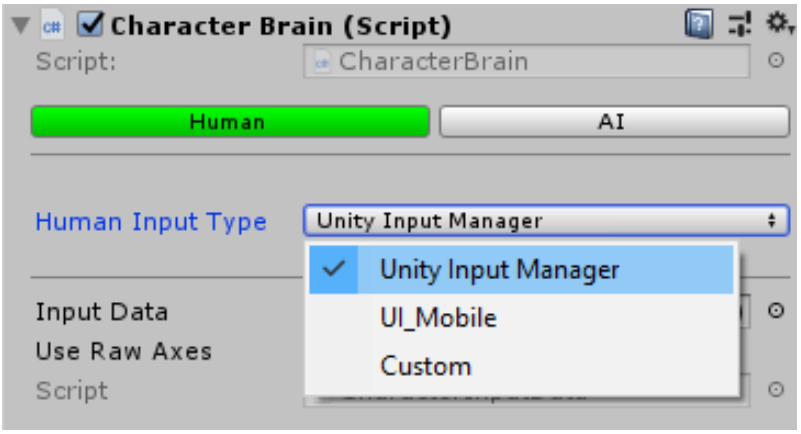


Figure 3.8: Available human input types.

3.4.1 Action

An action is a set of data (usually booleans and vectors) that simulate buttons and axes actions.

3.4.2 Human brain

Basically in a human brain the actions are updated using input devices (keyboard, mouse, joystick, UI, etc). Regardless of the input detection method used, all the actions must be previously defined using an *input data asset* (a ScriptableObject).

In order to update these actions an *input handler* is needed. This is a simple abstract component that needs to the implemented. It has the basic input functionalities used, such as *GetButton*, *GetButtonDown*, *GetButtonUp* and *GetAxis*. Each input handler should implement these methods in its own way.

The package contains two default input handler components, one for the classic *Unity's Input Manager* and another for the *Unity's UI* system (used in mobile). Additionally it supports a custom input handler mode, useful if you want to create your own handler.

These modes can be selected in the brain using the *Human Input Type* field.

Unity Input Manager This input handler reads inputs from the Unity's Input manager. Make sure the actions names (input data) and the axes from the input manager match exactly.

UI_Mobile This input handler reads all the mobile inputs components in the scene. This components are assigned to the UI elements responsible for converting UI Events into input values.

Custom A custom implementation of an input handler.

3.4.3 AI brain

In an AI brain the actions are determined by a script, based on the current behaviour type. There are two types of AI behaviours:

Sequence behaviour

Set of predefined actions stored as a *ScriptableObject*. Basically this behaviour tries to imitate a human player with scripted actions. The AI character will not be "smart" in any way.

Follow behaviour

This behaviour does a path calculation between the character an a target. In order to use this behaviour a *NavMesh* must be generated previously.

3.5 Character animation

A simple animation component that triggers animation clips based on the *CharacterMotor* state. It's compatible with the "Animator" controller, but it can be extended to other systems (for example the "Legacy" animation system) by deriving from this component class.

This component is probably the one that you will want to fully customize, since there is not a correct animation scheme or tool for the job. The package might include more tools in the future.

3.5.1 Blend trees

Blend trees allow us to blend between animation clips, simple as that. This is used in this implementation in two cases. The first one is when the character grounded and stable. The second one is when the character is not grounded.

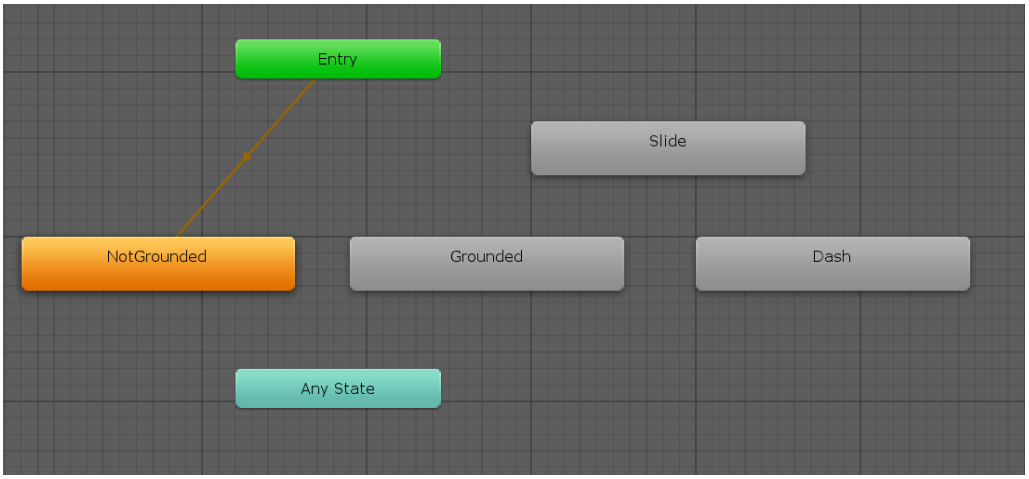


Figure 3.9: Character animator controller (base layer).

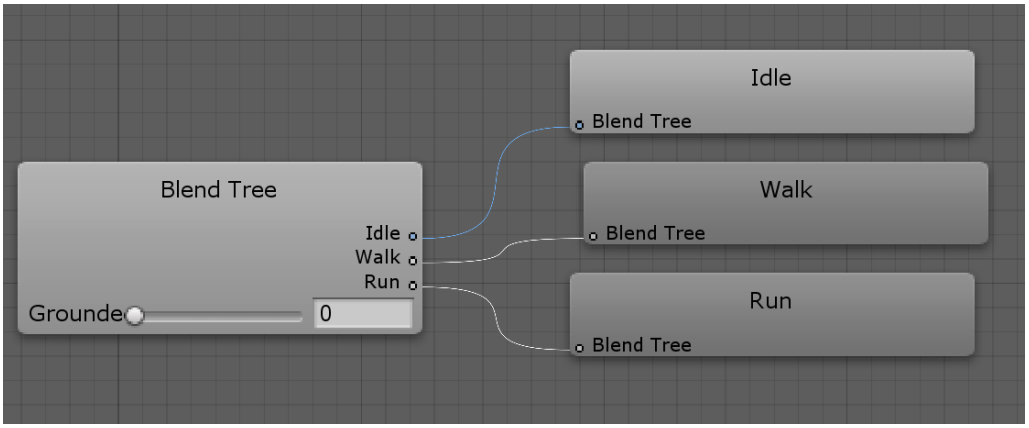


Figure 3.10: *Grounded* blend tree.

If you open the Animator controller included in the package, you will see something similar to figure 3.9. *NotGrounded* and *Grounded* are blend trees, the rest are simple *Animator* states with one clip in it.

A blend tree reads a specific variable to decide which clip of the tree it should play. This variable is fed to the blend tree by the *CharacterAnimation* component. Its name can be modified in the inspector.

For instance, in the *Grounded* blend tree the variable passed through is the current velocity magnitude. In figure 3.10 the blend tree structure is shown. We can see there are three clips in it: *Idle*, *Walk* and *Run*. The result will depend on the variable value, and the chosen threshold values as well.

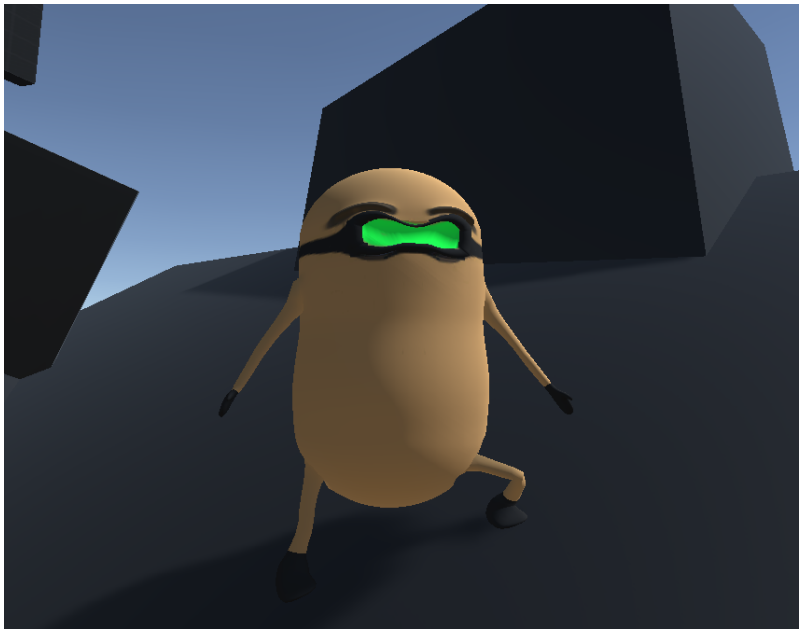


Figure 3.11: The demo character using IK foot placement.

3.5.2 IK foot placement

Although the character included with the package is not perfectly suited to work with IK foot placement (due to its proportions and the fact that it wasn't design for this task very well), the *CharacterAnimation* offers some basic functionality in this regard.

To make a character work with IK you'll need to:

1. Import an animated model using the *Humanoid* rig.
2. Configure its avatar (choose the left and right foot)
3. Configure the weight curves in the animated model (1 is full weight, 0 is no weight). Make sure the curves names match with the weight curves names from the *CharacterAnimation* component
4. Activate the *IK pass* in the Animator layer settings.
5. Activate the *IK Foot Placement* toggle in the *CharacterAnimation* component.

The IK function will cast a sphere towards the ground (for each foot) to determine where the foot needs to be placed at. It will also take into account the rotation as well, so the foot will be correctly rotated using the hit normal.

3.6 Character particles

The particles controller founded in the package is responsible for managing and triggering all the particles you see on screen, coming from the character. A particle prefab is specified as the main particle (its parameters will be modified by the controller).

All the particles are managed using a *ParticleSystem* pooler[3].

These particles are triggered in response to events. There are two types of events that the particles can listen to, animation events and *CharacterMotor* events.

3.6.1 Footsteps

For footstep animation events (defined in the animation import settings) are used, this is because they can be placed exactly at a certain time along the clip timeline. Figure 3.12 shows two animation events, for the *Walk* animation clip.

In the script we can add a public method and the animation system will call this methods for us. It's super important that the method name and the field *Function* from the animator settings match exactly. Otherwise the method will not be named at all.

3.6.2 *OnGroundedStateEnter* particles

If the character hits the ground it will produces some particles, whose velocity will vary depending on the falling speed at the moment of impact. To make this effect a *CharacterMotor* event was used, this event was the *OnGroundedStateEnter*. When this event is called the local vertical velocity is passed along as a parameter, which is great for this.

The magnitude obtained from the y component of the local vertical velocity is evaluated using a custom *AnimationCurve* (horizontal axis). The output of this curve (vertical axis) is used as the "start speed" of the *ParticleSystem* triggered.

3.7 Kinematic platforms

The package includes components that allow to script the movement of a platform in a particular way (depending on the component). Currently there are two types of kinematic platforms:

[3]A class that manages the visibility or activation of the objects it contains in a smart way. This is used to avoid the runtime instantiation of the objects (usually a prefab), thus improving performance.

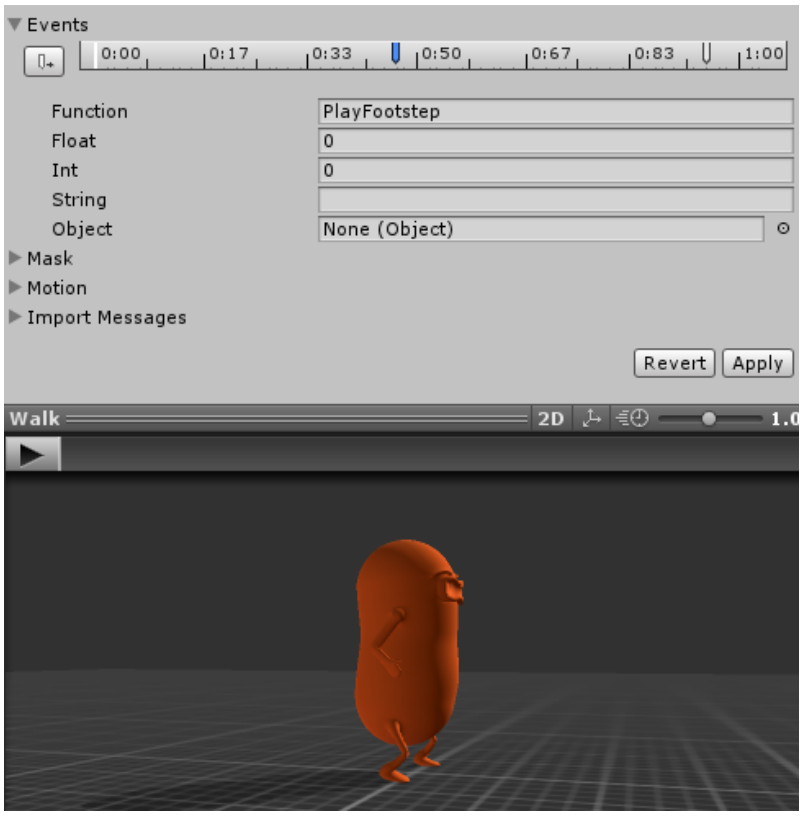


Figure 3.12: Footsteps animation events (import settings).

Node Based These platforms movement and rotation are purely based on nodes. Use this component to create platforms that moves and rotate precisely following a predefined path.

Action Based These platforms movement and rotation are defined by a single action for movement and rotation. Use this component if you want to create platforms with a pendulous nature, or infinite duration actions (for instance, if the platform should rotate forever).

3.8 Example : *NormalMovement* state

It is not necessary to create many states and transitions between them for creating a good playable character (the *NormalMovement* state is a good example of this). The *NormalMovement* state was created as a multi purpose state, it is responsible for basic grounded and not movement, gravity, walking and running, crouching, jumping, and so on.

This section is explained as a mini tutorial, basically justifying why was the *NormalMovement* state created that way. So you can extend from it, create your own version of it, or simply know more about the process behind.

If you want to know more in detail about this state please see the *"NormalMovement.cs"* script.

3.8.1 State controller

Since this is a state we must derive from the *CharacterState* class, implementing its abstract and virtual methods to define the behaviour we want (the same way you implement Unity's messages, like Start, Update, FixedUpdate, etc).

3.8.2 Basic state structure

In our vision we want to make the character move around and crouch, so we need to set its linear velocity and size properly to fit our needs. This need to be done in a frame by frame basis, so we proceed to write the update behaviour like this:

```
1  public override void UpdateBehaviour( float dt )
2  {
3    HandleSize( dt );
4    HandleMovement( dt );
5  }
6
```

HandleSize and *HandleMovement* will handle the size and movement respectively. The parameter "dt" is equals to *Time.deltaTime*.

3.8.3 Input handling

The more elemental thing to do is to read brain actions from the character (human or AI). This is accomplished by reading the *CharacterAction* struct directly.

For example to check if the *Jump* action was initiated (button pressed down):

```
1 if( characterBrain.CharacterAction.jumpPressed )
2 {
3   // Define the jump velocity vector ...
4 }
```

3.8.4 Size handling

Regarding the size, the only dimension that matters (in this case) is the height, so we define the *targetHeight* and implement the *HandleSize* method. Inside this method a targetSize is defined and passed to the *SetTargetBodySize* method.

```
1 void HandleSize( float dt )
2 {
3   // Define the targetHeight...
4
5   Vector3 targetSize = new Vector2( characterMotor.DefaultBodySize.
     x , targetHeight );
6
7   characterMotor.SetTargetBodySize( targetSize );
8 }
```

By doing this the character motor will check if the size is valid. If so, the character size will change when the *CharacterMotor* handle the size property. If not nothing is going to happen, the character size will remain the same.

3.8.5 Movement handling

One of the most important properties to set is the linear velocity in order to make the character move. So, calling the *CharacterMotor SetLinearVelocity* method would be a good place to start.

The final velocity will be the sum of three separated velocities. These are:

Controlled velocity Used for handling 100% controlled movement such as walk, run, air control movement, etc.

Vertical velocity Used for gravity-based movement such as jumping and gravity.

External velocity Used for handling external movement due to rigid-bodies impacts.

Now we need to implement the HandleMovement method:

```
void HandleMovement( float dt )
{
  // Set the values for all the velocities

  Vector3 velocity = controlledVelocity + verticalVelocity +
   externalVelocity;
  characterMotor.SetLinearVelocity( velocity );

}
```

3.8.6 Entering the state

By only implementing the *UpdateBehaviour* method we are reading a bunch of custom velocities (from the *NormalMovement* state exclusively) and combining them to obtain a final velocity vector. All these operations are done without previously knowing of the *CharacterMotor* linear velocity. If suddenly we are leaving another state and entering the *NormalMovement* state the result will be inconsistent, since the velocities values (the ones that define the new velocity) are not aware of the character linear velocity in that particular moment.

To fix this we can override the *EnterBehaviour* method, use it to read the current linear velocity vector and update our custom velocities properly.

```
public override void EnterBehaviour(float dt)
{
  verticalVelocity = Vector3.Project( characterMotor.LinearVelocity
    , transform.up );
  controlledVelocity = Vector3.ProjectOnPlane( characterMotor.
   LinearVelocity , transform.up );
  externalVelocity = Vector3.zero;

}
```

3.8.7 Collision events

We can take advantage of the character events to modify whatever parameter we want. In this case we need to convert the collision response into the *externalVelocity*

vector.

Another thing to do is to make zero our vertical velocity when the character hits something with its head. We achieve this by "listening" to the *OnHeadHit* and *OnContactHit* events, using the *OnHeadHit* and *OnContactHit* methods respectively. This is the code used:

```
1  void OnEnable()
2  {
3    characterMotor.OnHeadHit += OnHeadHit;
4    characterMotor.OnContactHit += OnContactHit;
5  }
6
7  void OnDisable()
8  {
9    characterMotor.OnHeadHit -= OnHeadHit;
10   characterMotor.OnContactHit -= OnContactHit;
11 }
12
13 void OnHeadHit( CollisionInfo collisionInfo )
14 {
15   verticalVelocity = Vector3.zero;
16 }
17
18
19 void OnContactHit( Vector3 impulse )
20 {
21   if( !rigidbodyResponseParameters.reactToRigidbodies )
22     return;
23
24   externalVelocity = impulse * rigidbodyResponseParameters.
     impulseMultiplier;
25 }
```